Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Triennale in Matematica

ISOMORFISMO TRA ALBERI
Algoritmi e Complessità computazionale

Tesi di Laurea in Informatica Teorica

Relatore:
Chiar.mo Prof.
Simone Martini

Presentata da:
Ezio Catelli

III Sessione
Anno Accademico 2012-2013

Indice

Introduzione **iii**

1 Modelli di computazione **1**

1.1 Macchine di Turing 1

1.2 Circuiti Booleani 4

2 Classi di complessità **7**

2.1 Spazio e tempo 7

2.1.1 Tempo 7

2.1.2 Spazio 10

2.2 Classi di complessità 11

2.2.1 Riduzioni 13

2.2.2 **L** è contenuto in **P** 20

3 GI nel caso di alberi **23**

3.1 Isomorfismo tra Alberi 23

3.2 L'algoritmo 25

3.3 La complessità computazionale 30

Bibliografia **33**

Introduzione

L'argomento trattato nella tesi riguarda il problema (decisionale) di determinare l'esistenza di un isomorfismo tra due alberi dati (GI). Esso consiste nel decidere se esiste una mappa bigettiva tra l'insieme dei vertici di un dato grafo che preserva gli archi, ovvero che preserva la relazione di adiacenza tra coppie di vertici.
Nel primo e secondo capitolo si richiamano i prerequisiti fondamentali della materia in argomento. In essa sono esposti i modelli astratti usati per studiare il problema in questione; il principale è la Macchina di Turing. La definizione di classe di complessità, nel caso di **P** e **NP**, è intimamente legata a quella di Macchina di Turing. A una stessa classe di complessità possono appartenere più problemi. Tramite le classi di complessità classifichiamo i problemi; in particolare notiamo che il problema di isomorfismo tra grafi ad ora non è ancora stato dimostrato essere decidibile in tempo polinomiale (di essere in **P**) né di essere il più difficile dei problemi non decidibili in tempo polinomiale (di essere **NP-completo**). A meno di considerare suoi casi particolari, come spiego nel terzo capitolo, GI non è ancora stato scoperto appartenere a una classe computazionale piccola (**P** o la classe dei problemi **NP-completi**), rispetto a quella più generale dei problemi risolvibili tramite Macchine di Turing non deterministiche operanti in tempo polinomiale (la classe **NP**). Inoltre presento, per completezza, un modello di computazione distinto dalla Macchina di Turing che è il Circuito Booleano. Esso permette di risolvere GI, nel caso di alberi, tramite calcolatori in parallelo con un numero polinomiale di processori in tempo logartimico (la classe $\mathbf{NC}_1$).

Nel terzo capitolo si presenta, seguendo i passi della dimostrazione compiuta da Lindell, che l'isomorfismo tra grafi nel caso di alberi è risolubile in spazio logaritmico. Si vedrà una relazione di pre-ordine sull'insieme degli alberi, per la quale si ottiene una condizione necessaria e sufficiente di isomorfismo tra alberi, e con essa l'algoritmo.

Infine devo ringraziare, per il punto d'arrivo raggiunto con la preparazione di questo lavoro, la mia famiglia, il professor Martini, Saverio Tassinari, Davide Maccarrone e chi mi ha dato supporto, nessuno escluso.

Capitolo 1

Modelli di computazione

1.1 Macchine di Turing

Vediamo due differenti modelli di computazione sequenziale: la Macchina di Turing deterministica e la Macchina di Turing non deterministica. Sono il modello usato per simulare la computazione svolta da un calcolatore. Presentiamo ora la definizione di Macchina di Turing deterministica:

Definizione 1.1. Una k-Macchina di Turing deterministica è una quadrupla (K, Σ, δ, s) e un numero naturale k. Ove K è un insieme finito di stati in cui si trova la macchina durante la computazione, Σ è un insieme finito di caratteri scritti (l'alfabeto dellla macchina) sui nastri e k è il numero di nastri (su cui opera la macchina attraverso un cursore). Il primo stato iniziale in cui si trova la macchina è rappresentato da $s \in K$ e δ è la funzione di transizione della macchina tra uno stato e l'altro, ossia una mappa da $K \times \Sigma^k$ a $(K \cup \{h, SI, NO\}) \times (\Sigma \times \{\leftarrow, \rightarrow, \bullet\})^k$. Tutte le stringhe sui loro rispettivi nastri iniziano al primo carattere con il simbolo $\triangleright \in \Sigma$; in più per denotare uno spazio su un nastro usiamo il carattere $\sqcup \in \Sigma$. Il primo nastro contiene la stringa che è il messaggio in ingresso della k-Macchina di Turing. L'ultimo nastro (il k-esimo) contiene la stringa che è il messaggio in uscita della Macchina di Turing. Se è presente solo il primo nastro abbiamo una Macchina di Turing.

Osservazione 1. Consideriamo per evitare ambiguità che K e Σ siano insiemi disgiunti. E che h (stato di arresto), NO (stato di rigetto del messaggio in input), SI (stato di accettazione del messaggio), $\leftarrow$ (direzione del cursore che legge il nastro), $\rightarrow$ (direzione del cursore), $\bullet$ (attesa del cursore) non appartengano a $K \cup \Sigma$.

Andiamo a illustrare più chiaramente con un esempio la definizione appena annunciata:

Esempio 1.1. Costruiamo una 2-Macchina di Turing deterministica M. Specifichiamo in una tabella la funzione di transizione δ:

$p \in K\ \sigma_1 \in \Sigma\ \sigma_2 \in \Sigma$	$\delta(p, \sigma_1, \sigma_2)$
$s\ 0\ \sqcup$	$(s, 0, \rightarrow, 0, \rightarrow)$
$s\ 1\ \sqcup$	$(s, 1, \rightarrow, 1, \rightarrow)$
$s\ \triangleright\ \triangleright$	$(s, \triangleright, \rightarrow, \triangleright, \rightarrow)$
$s\ \sqcup\ \sqcup$	$(q, \sqcup, \leftarrow, \sqcup, \bullet)$
$q\ 0\ \sqcup$	$(q, 0, \leftarrow, \sqcup, \bullet)$
$q\ 1\ \sqcup$	$(q, 1, \leftarrow, \sqcup, \bullet)$
$q\ \triangleright\ \sqcup$	$(p, \triangleright, \rightarrow, \sqcup, \leftarrow)$
$p\ 0\ 0$	$(p, 0, \rightarrow, \sqcup, \leftarrow)$
$p\ 1\ 1$	$(p, 1, \rightarrow, \sqcup, \leftarrow)$
$p\ 0\ 1$	$(NO, 0, \bullet, 1, \bullet)$
$p\ 1\ 0$	$(NO, 1, \bullet, 0, \bullet)$
$p\ \sqcup\ \triangleright$	$(SI, \sqcup, \bullet, \triangleright, \rightarrow)$

La computazione è omessa (la computazione determina se una stringa è palindroma).
Possiamo vedere la funzione δ come il programma vero e proprio della 2-Macchina di Turing M. Essa specifica, per ogni combinazione di stato corrente $q \in K$ e simboli correnti $(\sigma_1, \sigma_2) \in (\Sigma \times \Sigma)$ durante la computazione la quintupla $\delta(q, \sigma_1, \sigma_2) = (p, \rho_1, D_1, \sigma_2, D_2)$. Ove p è lo stato seguente allo stato corrente, ρ_1 e ρ_2 sono i simboli da sovrascrivere sui rispettivi simboli σ_1 e σ_2 sui rispettivi nastri; $(D_1, D_2) \in \{\leftarrow, \rightarrow, \bullet\}^2$ è la direzione nella quale i

cursori del primo e secondo nastro devono muoversi. Si ha infine che $\triangleright$ dirige sempre il cursore del primo e secondo nastro a destra, e non è mai eliminato, nel senso seguente: $\delta(q, \triangleright, \triangleright) = (p, \triangleright, \rightarrow, \triangleright, \rightarrow)$. Vediamo come funziona il programma eseguito dalla 2-Macchina di Turing M nel suo complesso: all'inizio lo stato è s, le stringhe sui nastri sono inizializzate a $\triangleright$ seguite da una stringa di lunghezza finita di simboli $x \in (\Sigma \setminus \{\sqcup\})^*$[1]; i cursori puntano sul primo simbolo, cioè $\triangleright$. Chiamiamo x messaggio in ingresso della 2-Macchina di Turing, la stringa scritta sul primo nastro.
Dalla configurazione iniziale di *stato della Macchina M/posizione del cursore lungo i nastri* si svolge un passo secondo la definizione di δ arrivando a un'altra configurazione *cambiando stati/muovendo i cursori lungo i nastri* e sovrascrivendo i simboli opportunamente in accordo a δ.
Diciamo che la Macchina M termina la computazione nel caso uno dei tre stati h, SI, NO sia raggiunto. In particolare, se la computazione termina nello stato SI diciamo che è accettato il messaggio in ingresso, se NO che lo rigetta. Possiamo definire $M(x)$ il messaggio in uscita della Macchina M come la stringa ottenuta sull'ultimo nastro (k-esimo nastro) al raggiungimento dello stato h. Altrimenti $M(x) = SI$ o $M(x) = NO$.
È possibile che la macchina M non raggiunga mai uno stato in cui termina. In tal caso si pone $M(x) \nearrow$.

Diamo ora la definizione di Macchina di Turing non deterministica. La peculiarità sua propria è di poter svolgere più computazioni simultaneamente per il fatto che δ non sarà una semplice funzione, ma una relazione Δ. Questo tipo differente di Macchina di Turing ha un modello di costo più efficiente rispetto a quella esposta nella definizione precedente (nel senso del seguente Teorema 2.1.1).

[1] N.B.: L'asterisco ad apice significa l'insieme di tutte le possibili stringhe componibili con i simboli appartenenti all'insieme precedente l'asterisco

Definizione 1.2. Una Macchina di Turing non deterministica N è una quadrupla (K, Σ, Δ, s) ove K, Σ e s sono come definiti precedentemente. Δ è una relazione su $(K \times \Sigma) \times [(K \cup \{h, SI, NO\}) \times \Sigma \times (\leftarrow, \rightarrow, \bullet)]$. Abbiamo una relazione (non più una funzione) che regola la transizione tra uno stato e lo stato della computazione successiva. Consideriamo ogni passo della computazione come un cambiamento dalla configurazione iniziale di *stato della Macchina/ posizione del cursore lungo i nastri* secondo la definizione di Δ in un'altra configurazione *cambiando stati/ muovendo i cursori lungo i nastri* e sovrascrivendo i simboli opportunamente. Abbiamo anche qui un primo nastro contenente la stringa x che è il messaggio in ingresso della Macchina di Turing. Se tra le computazioni non deterministiche che si sono svolte ce n'è almeno una che termina nello stato SI, diciamo che la Macchina termina la computazione e accetta il messaggio in ingresso. Un messaggio in ingresso x è rigettato dalla Macchina di Turing non deterministica solo se tutte le computazioni che si sono svolte sui vari nastri necessari allo svolgimento non accettano il messaggio in ingresso. È possibile che la macchina non raggiunga mai uno stato in cui termina. In tal caso si pone $N(x) \nearrow$.

1.2 Circuiti Booleani

Vediamo un modello di computazione in parallelo. Il modello usato per simulare la computazione svolta da più calcolatori è il Circuito Booleano. Presentiamo ora la definizione di Circuito Booleano:

Definizione 1.3. Un Circuito Booleano C è un grafo orientato $C = (V, E)$ dove i vertici $V = \{1, 2, ..., n\}$ sono le porte logiche del Circuito Booleano C. Devono valere le seguenti proprietà:

- $\forall (i, j) \in E \Rightarrow i < j$
- $\forall\ i \in V$ il massimo numero di archi entranti in un vertice è espresso da una variabile $deg = 0, 1, 2$ tale che:

$deg = 0$ identifica i nodi di ingresso per il Circuito Booleano C, essi vengono rappresentati con un'etichetta $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, ...\}$

$deg = 1$ identifica nodi rappresentati con l'etichetta $s(i) = \neg$

$deg = 2$ identifica nodi rappresentati con un'etichetta $s(i) \in \{\wedge, \vee\}$

La porta logica $n \in V$ viene detta nodo di uscita per il Circuito Booleano C. Dato T un assegnamento di valori di verità alle variabili $\{x_1, x_2, ..., x_n\}$ diciamo che il valore di verità del Circuito Booleano C è $T(n)$, ove l'assegnamento applicato ai nodi è definito come:

- $T(i) = T(x_i)$ se i ha $deg = 0$
- $T(i) = \neg\, T(i')$ se i ha $deg = 1$ con i' tale che $(i', i) \in E$ e $s(i) = \neg$
- $T(i) = T(i') \wedge T(i'')$ oppure $T(i) = T(i') \vee T(i'')$ se i ha $deg = 2$ con i', i'' tali che $(i', i), (i'', i) \in E$ e $s(i) \in \{\wedge, \vee\}$

La *dimensione* del Circuito Booleano è il numero totale dei nodi presenti in esso.

Per parlare di messaggio in ingresso per un Circuito Booleano la definizione sopra non basta. Nel senso che $|x|$ del messaggio in ingresso non può essere limitata dal numero di nodi di ingresso per un dato Circuito Booleano. Si rende necessaria la seguente:

Definizione 1.4. Una *famiglia di circuiti* è una sequenza infinita $(C_0, C_1, ...)$ di Circuiti Booleani dove il numero a pedice denota il numero di nodi di ingresso del rispettivo circuito.

Capitolo 2

Classi di complessità

Prima di enunciare le principali classi di complessità che andremo a usare più avanti, chiariamo quali sono le risorse utilizzate da un algoritmo operante secondo uno dei modelli di computazione visto sopra.

2.1 Spazio e tempo

2.1.1 Tempo

Consideriamo Macchine di Turing che terminano sempre. Consideriamo le risorse fisiche necessarie ad una Macchina di Turing M deterministica per realizzare una computazione:

Definizione 2.1. Il *tempo* richiesto da M operante sul messaggio in ingresso x è il numero di passi t necessari alla Macchina per raggiungere uno stato di terminazione.
Sia f una funzione dagli interi non negativi in sè. La Macchina di Turing M opera in *tempo* $t = f(n)$ se $\forall$ x messaggio in ingresso, il *tempo* richiesto da M per svolgere la computazione fino a uno stato di terminazione è al più $f(|x|)$, ove $|x|$ è la lunghezza (i.e. il numero di caratteri) del messaggio in ingresso. Diciamo che $L \subseteq (\Sigma \setminus \{\sqcup\})^*$ è deciso in *tempo* $f(n)$ dalla Macchina di Turing M, ove

$$L = \{x \mid M(x) = SI\}$$

e M è la Macchina di Turing in questione operante in *tempo* $f(n)$. In questo caso scriviamo $L \in$ **Tempo(f(n))**, e L viene detto linguaggio. **Tempo(f(n))** la chiamiamo classe di complessità; si tratta di un insieme di linguaggi L ognuno dipendente da una particolare Macchina di Turing che lo accetta entro il *tempo* richiesto.

Consideriamo le risorse fisiche necessarie ad una Macchina di Turing N non deterministica per realizzare una computazione:

Definizione 2.2. Il *ntempo* richiesto da N operante sul messaggio in ingresso x è il numero di passi t necessari alla Macchina per raggiungere uno stato di terminazione in una delle computazioni svolte su uno dei vari nastri necessari allo svolgimento.
Sia f una funzione dall'insieme degli interi non negativi in sè. La Macchina di Turing N opera in *ntempo* $t = f(n)$ se $\forall\ x$ messaggio in ingresso, il *ntempo* richiesto da N per svolgere la computazione fino a uno stato di termine è al più $f(|x|)$, ove $|x|$ è la lunghezza nel numero di caratteri del messaggio in ingresso. Diciamo che $L \subset (\Sigma \setminus \{\sqcup\})^*$ è deciso in *ntempo* $f(n)$ dalla Macchina di Turing N, ove

$$L = \{x \mid N(x) = SI\}.$$

E in questo caso scriviamo $L \in$ **NTempo(f(n))**, e L viene detto linguaggio.

Osservazione 2. Il caso in cui la Macchina di Turing deterministica (non deterministica) non termina la computazione è perché supponiamo impieghi un numero di passi ∞ per giungere al termine della (di almeno una delle) computazione(computazioni). A priori non è detto di sapere quando la Macchina di Turing non termina effettivamente. È rilevante notare che il problema di stabilire se una data Macchina di Turing terminerà per un fissato messaggio in ingresso x (i.e. Halting Problem) è indecidibile, ossia non esiste un algoritmo (una Macchina di Turing) capace di risolverlo (cioè che termini in uno stato h, SI o NO).

Teorema 2.1.1. *Supponiamo che un linguaggio L sia deciso da una Macchina di Turing non deterministica N in tempo $f(n)$.*
Allora L è deciso da una 3-Macchina di Turing deterministica in tempo $\boldsymbol{O}(c^{f(n)})$, con $c > 1$ costante dipendente da N.

Dimostrazione. Sia N una Macchina di Turing non deterministica. Consideriamo $\forall(q, \sigma) \in K \times \Sigma$ l'insieme delle scelte deterministiche:

$$C_{q,\sigma} = \{(q', \sigma', D) \mid ((q, \sigma), (q', \sigma', D)) \in \Delta\},$$

esso è finito (se fosse infinito avrei che K o Σ o l'insieme dei movimenti del cursore sarebbe infinito, ciò è assurdo). Perciò prendiamo $d = max_{q,\sigma}|C_{q,\sigma}|$ (sarà $d > 1$). Facciamo vedere che N si può simulare da un algoritmo deterministico. Innanzitutto vediamo una sequenza di t scelte non deterministiche di N come una sequenza $(c_1, c_2, ..., c_t)$ di t interi $\in \{0, 1, ..., d-1\}$ (i.e. un numero di t cifre in base d). Costruiamo M Macchina di Turing che le considera tutte, in ordine crescente di lunghezza t della sequenza $(c_1, c_2, ..., c_t)$. Per ogni passo M è dotata di un nastro in cui simula la computazione che N avrebbe prendendo la scelta deterministica c_i al passo i per i primi t passi, partendo ogni volta dal messaggio in ingresso x per N; su di un secondo invece mantiene memorizzata la sequenza di t interi. Ora, se entro t passi N non raggiunge uno stato di terminazione SI, nemmeno M termina. Quindi si procede ricalcolando un'altra sequenza di numeri $\in \{0, 1, ..., d-1\}$. E si presentano i seguenti casi:

- presto o tardi si arriva a uno stato di terminazione SI;
- M trova tra una scelta simulata di N e la successiva un passaggio di configurazioni $\notin \Delta$, ovvero N ha appena raggiunto uno stato di terminazione NO o h

Chiaramente M in tutto questo procederà in un numero di passi al più

$$\Sigma_{t=1}^{f(n)} d^t = \boldsymbol{O}(d^{f(n)+1})[1]$$

[1]N.B.: t è limitato superiormente da $f(n)$ dato che la complessità di N è $f(n)$

. □

Consideriamo il tempo necessario ad una famiglia di circuiti per realizzare una computazione:

Definizione 2.3. Il *tempo parallelo* richiesto da $(C_0, C_1, ...)$ operante sul messaggio in ingresso x è il numero di nodi t del più lungo percorso tra quelli nei grafi C_1, C_2, ...
Sia f una funzione dall'insieme degli interi non negativi in sè. La famiglia di circuiti opera in *tempo parallelo* $t = f(X)$ se $\forall$ n la profondità (lunghezza del più lungo percorso in un grafo) di C_n è al più $f(X)$. Diciamo che $L \subset \{\mathbf{true}, \mathbf{false}\}^*$ è deciso in *tempo parallelo* $f(X)$ dalla famiglia di circuiti $(C_0, C_1, ...)$, ove

$$L = \{x \mid \text{il messaggio in uscita di } C_{|x|} \text{ è il valore di verità } \mathbf{true}\}.$$

2.1.2 Spazio

Diamo la definizione di spazio utilizzato da una k-Macchina di Turing:

Definizione 2.4. Lo spazio richiesto dalla k-Macchina di Turing operante sul messaggio in ingresso x è $\Sigma_{i=1}^{k}|y_i|$ dove y_i sono le stringhe posizionate sui rispettivi nastri al termine della computazione. La k-Macchina di Turing opera in spazio $f(n)$ se $\forall$ x messaggio in ingresso, lo spazio richiesto per svolgere la computazione fino a uno stato di termine è al più $f(|x|)$, ove $|x|$ è il numero di caratteri del messaggio in ingresso. Diciamo che $L \subset (\Sigma \setminus \{\sqcup\})^*$ è deciso in spazio $f(n)$ dalla k-Macchina di Turing, ove $L = \{x \mid N(x) = SI\}$.

Osservazione 3. La stima usata nella definizione sopra per lo spazio impiegato è una sovrastima dato che la lunghezza del messaggio in ingresso, e in uscita, è sempre conteggiata nell'ammontare totale.

Diamo la definizione di quantità di lavoro compiuto da una famiglia di circuiti $(C_0, C_1, ...)$:

Definizione 2.5. La quantità di lavoro svolta dalla famiglia di circuiti operante sul messaggio in ingresso x è il valore maggiore tra le dimensioni di C_1, C_2, ... Sia g una funzione dall'insieme degli interi non negativi in sè. La famiglia di circuiti opera in quantità di lavoro $g(X)$ se $\forall$ n la dimensione di C_n è al più $g(|X|)$. Il messaggio in ingresso di una famiglia di circuiti $(C_0, C_1, ...)$ è una stringa x di caratteri in $\{\mathbf{true}, \mathbf{false}\}^*$[2]. Diciamo che la famiglia di circuiti accetta x, il messaggio in ingresso, se il nodo di uscita di $C_{|x|}$ ha valore booleano **true**. Similmente diciamo che accetta un linguaggio L se accetta ogni messaggio in ingresso $x \in L$.

Al contrario di quanto visto nell'Osservazione 2 qui abbiamo per una famiglia di circuiti booleani il seguente Teorema 2.1.2:

Teorema 2.1.2. *$\exists$ L un linguaggio indecidibile accettato da una famiglia di circuiti tale che la dimensione di C_n è al più polinomiale in n.*

Dimostrazione. Vedi 2.2.1 □

Osservazione 4. D'ora in avanti useremo come sinonimi le parole problema, algoritmo, linguaggio: per un dato problema possiamo costruire una codifica concisa e de-codificabile in modo da trasformarlo in un linguaggio in grado di essere elaborato da un modello di computazione (tra quelli visti sopra), cioè un algoritmo capace di raggiungere uno stato di terminazione (ad esempio di accettazione o rigetto).

2.2 Classi di complessità

Il problema di isomorfismo tra grafi (GI) ad ora si risolve con algoritmi non facili. Nel senso che non è ancora dimostrata l'appartenenza del problema alla classe **P** seguente:

Definizione 2.6. Sia L un linguaggio deciso da una Macchina di Turing deterministica in tempo polinomiale, ovvero $L \in \mathbf{Tempo}(n^k)$, con k numero reale. Allora $\mathbf{P} := \bigcup_{k=0}^{n} \mathbf{Tempo}(n^k)$.

[2] $\{\cdot\}^*$ indica l'insieme di tutte le stringhe formate con i caratteri dell'insieme.

Riconoscendo che una Macchina di Turing deterministica è un caso particolare di Macchina di Turing non deterministica, anche nel caso particolare del modello di calcolo più efficiente (vedi Teorema 2.1.1), è immediato verificare l'appartenenza di GI alla classe **NP** definita come segue:

Definizione 2.7. Sia L un linguaggio deciso da una Macchina di Turing non deterministica in tempo polinomiale, ovvero $L \in \mathbf{NTempo}(n^k)$, con k numero reale. Allora $\mathbf{NP} := \bigcup_{k=0}^{n} \mathbf{NTempo}(n^k)$.

Nel Capitolo 3 discuteremo come risolvere il problema di isomorfismo tra alberi in un caso speciale, all'interno della classe di complessità **L**, definita come segue:

Definizione 2.8. Sia L un linguaggio deciso da una k-Macchina di Turing deterministica in spazio logaritmico. Allora $\mathbf{L} := \bigcup_{L \in \mathbf{Spazio}(\log(n))} L$.

Analogamente a **NP** si definisce **NL**. Abbiamo inoltre la seguente:

Definizione 2.9. Sia L un linguaggio deciso da una Macchina di Turing non deterministica in tempo logaritmico, ovvero $L \in \mathbf{NSpazio}(log(n))$, e tale che $\forall$ x messaggio in ingresso $\exists$ al più una computazione tra le varie necessarie allo svolgimento che termini in uno stato di accettazione. Allora $\mathbf{UL} := \mathbf{NSpazio}(log(n))$.

Infine definiamo una classe di complessità per una famiglia di circuiti:

Definizione 2.10. Sia $L \subseteq \{\mathbf{true}, \mathbf{false}\}^*$ un linguaggio deciso da una famiglia di circuiti $(C_0, C_1, ...)$ in tempo parallelo $\boldsymbol{O}(f(n))$ e quantità di lavoro $\boldsymbol{O}(g(n))$, ove $f(n)$ e $g(n)$ sono funzioni dall'insieme degli interi non negativi in sè. Diciamo in questo caso che $L \in \mathbf{PT/WK}(f(n), g(n))$. Allora

$$\mathbf{NC}_j := \{L \mid L \in \mathbf{PT/WK}(\log^j(n), n^k)\}$$

Inoltre

Definizione 2.11.

$$\mathbf{NC} := \bigcup_{k \in \mathbb{N}} \{L \mid L \in \mathbf{PT/WK}(\log^k(n), n^k)\}$$

2.2.1 Riduzioni

Un concetto fondamentale e potente per trattare la teoria è quello di riduzione tra linguaggi. Nel senso che un problema codificato in linguaggio, linguaggio per cui esiste un dispositivo in grado di riconoscerlo (i.e. una Macchina di Turing o una famiglia di circuiti che termina la sua esecuzione) viene trasformato in un altro utilizzando una certa quantità di risorse. Per rendere la riduzione efficace nel distinguere tra loro problemi anche molto simili, richiediamo il minor costo di computazione possibile, nel senso della Definizione 2.12.

Osservazione 5. Diciamo che la Macchina di Turing deterministica M computa la funzione $f : \Sigma^* \to \Sigma^*$ se: $\forall$ stringa $x \in \Sigma^* \Rightarrow M(x) = f(x)$.

Definizione 2.12. Siano L_1 e L_2 linguaggi, diciamo che L_1 si riduce a L_2 (i.e. $L_1 \propto L_2$) se $\exists$ una funzione $R : \Sigma^* \to \Sigma^*$ computabile da una Macchina di Turing deterministica in spazio $\boldsymbol{O}$ $(\log(n))$ tale che $\forall$ x messaggio in ingresso vale:

- $x \in L_1 \Leftrightarrow R(x) \in L_2$

Proposizione 2.2.1. Se R è una riduzione da L_1 verso L_2 e R' è una riduzione da L_2 verso L_3

$$\Rightarrow L_1 \propto L_3.$$

Dimostrazione. Sia $x \in L_1$, allora $R'(R(x)) \in L_3$ per come sono definite le due riduzioni R ed R'. Mostriamo che $R' \circ R$ è computabile da una Macchina di Turing deterministica in spazio $\boldsymbol{O}(\log n)$, sia questa M la composizione delle due Macchine di Turing M_R e $M_{R'}$. Bisogna verificare che i messaggi in uscita di M_R (messaggi d'ingresso per $M_{R'}$) siano opportuni per un utilizzo complessivo delle risorse in $\boldsymbol{O}(\log n)$ spazio; procediamo quindi nel costruire M come segue. Nel simulare $M_{R'}$ sul messaggio in ingresso $R(x)$ memorizziamo ogni volta la posizione i del cursore sul nastro dedicato al messaggio in ingresso di $M_{R'}$ (che è il messaggio in uscita di M_R). In questo modo non

abbiamo bisogno di un nastro di M per memorizzare $R(x)$, che può essere più che logaritmico nel numero di caratteri! Per fare questo conveniamo che M abbia un nastro apposito per la memorizzazione di i. All'inizio $i = 1$, e abbiamo che M è in procinto di iniziare la simulazione di M_R sul messaggio in ingresso x. Ricordiamo che il primo passo di M nella computazione simulata di $M_{R'}$ consiste nel cursore che legge sul messaggio in ingresso il primo simbolo $\triangleright$ scritto a sinistra. Successivamente, ogni spostamento a destra del cursore conveniamo che provoca un incremento unitario di i e l'esecuzione della successiva computazione di M_R sul messaggio in ingresso x fino alla prima successiva uscita di un nuovo simbolo nel messaggio in uscita di M_R. Quindi M scansiona questo nuovo simbolo per la simulazione di $M_{R'}$, e così via. Nel caso che il cursore per il messaggio in ingresso di $M_{R'}$ rimanga fermo memorizziamo il simbolo in ingresso scansionato per un possibile utilizzo nella computazione successiva. Nel caso che il cursore per il messaggio in ingresso di $M_{R'}$ si muova a sinistra decrementiamo i di 1, e quindi si esegue M_R sul messaggio in ingresso x dall'inizio contando in un altro nastro i simboli usciti fin d'ora e fermando il conto quando l'i-esimo (i.e. l'$i-1$-esimo) simbolo esce sul messaggio in uscita di M_R. A questo punto la simulazione di $M_{R'}$ viene ripresa. Questa computazione di M quindi richiede, per ipotesi, $\boldsymbol{O}(\log n)$ spazio per le due computazioni di M_R e $M_{R'}$. E infine $\boldsymbol{O}(\log n)$ spazio per la memorizzazione di i dato che (vedi [1]) la lunghezza del messaggio in uscita $R(x)$ di M_R è al più polinomiale in $n = |x|$, e quindi i è costituito da al più $\log(n)$ caratteri. □

Osservazione 6. Abbiamo quindi in virtù della Definizione 2.12 che si individuano classi di probemi in cui possiamo eleggere un rappresentante verso il quale si riducono tutti gli altri problemi appartenenti alla medesima classe. Ove si considera la relazione di preordine individuata dalla riducibilità (la transitività della riducibilità è dimostrata nella Proposizione 2.2.1).

Dimostrazione. 2.1.2 Supponiamo sia $L \subseteq \{\mathbf{true}, \mathbf{false}\}^*$ (i.e. $\{1, 0\}^*$) un linguaggio indecidibile espresso in un alfabeto di soli due simboli. Poniamo

$U = \{\mathbf{true}^n \mid n$ scritto in base 2 (i.e. n_2), ove $n_2 \in L\}$ (i.e. 1^n) linguaggio fatto di un solo simbolo **true** (i.e. 1); L si riduce in tempo esponenziale a U indecidibile, essendo L indecidibile. Vediamo che U è deciso da una famiglia $(C_1, C_2, ...)$ di circuiti, polinomiali nella dimensione in ingresso, così costruita:

$$C_n = \begin{cases} \text{ha } n-1 \text{ nodi di etichetta } s(\cdot\,) = \wedge \\ \text{(i.e. prendiamo la congiunzione} \\ \text{degli } n \text{ nodi di input } \overbrace{\mathbf{truetrue...true}}^{n} \\ \text{(i.e. } \underbrace{11....1}_{n}\text{))} & \text{se } \mathbf{true}^n \in U \\ \\ \text{ha tutti i nodi di ingresso e un solo nodo per} \\ \text{il messaggio in uscita impostato a } \mathbf{false} & \text{se } \mathbf{true}^n \notin U \end{cases}$$

□

Osservazione 7. Consideriamo il preordine indotto da $\propto$, allora l'elemento massimale (rispetto a quest'ordine) dell'insieme degli elementi di una data classe di complessità C viene detto C-completo.

Vediamo ora il primo problema completo (i.e. un rappresentante della sua classe di equivalenza in base alla relazione d'equivalenza $\propto$) per **NP**, in ordine cronologio, Satisfiability:

Teorema 2.2.2. *(Cook) Soddisfattibilità è **NP**-completo.*

Dimostrazione. Per la dimostrazione consideriamo la riduzione non in spazio logaritmico bensì in tempo polinomiale. SAT (i.e. satisfiability) appartiene a **NP**, un algoritmo non deterministico è dato da una soluzione (un assegnamento dei valori di verità alle variabili che soddisfa l'espressione logica di SAT) e un controllo (in tempo polinomiale) della validità di questa soluzione. Conveniamo d'ora in avanti che l'espressione di SAT da soddisfare sia scritta in forma normale congiuntiva (i.e. FNC). Rimane ora da far vedere

che $L \propto L_{SAT}$ per ogni $L \in \mathbf{NP}$. Quindi, supponiamo $L \in \mathbf{NP}$. Allora L è deciso da una Macchina di Turing M in tempo non deterministico $p(n)$, ove p è un polinomio. Sia Σ l'insieme dei simboli usati da M. Mostriamo che da $x \in L \subset$, in Σ^*, possiamo arrivare tramite una riduzione in tempo polinomiale f_L ad un'istanza di SAT: se $x \in \Sigma^*$ messaggio in ingresso per M, allora (vedi [2]) il numero di passi necessari alla computazione di M, e la lunghezza della soluzione trovata che fa terminare M con risposta SI, sono limitati superiormente da $p(n)$, ove $n = |x|$. Quindi i nastri usati da M coinvolgeranno al più $p(n)$ caselle siccome il cursore si muove di al più una casella in un singolo passo. Allora lo stato della computazione ad ogni istante può essere specificato esaurientemente dal contenuto della casella corrente, dallo stato $q \in K \cup \{h, SI, NO\}$ corrente e dalla posizione j del cursore lungo il nastro (dovremmo parlare di più nastri, ma per chiarezza ci limitiamo al caso di uno solo, al più ve ne sono un numero costante (dipendente soltanto da M) e finito). E quindi, tramite f_L, un numero finito di espressioni booleane con un numero finito di letterali concatenati in un'unica espressione per SAT. Infatti il limite polinomiale della computazione ci consente di descrivere la computazione completamente usando un numero finito di variabili booleane e un'espressione logica opportuna per esse, nel seguente modo: etichettiamo gli elementi di $K \cup \{h, SI, NO\}$ come q_0, $q_1 = SI$, $q_2 = NO$, q_3, ..., q_r, ove $r = |K \cup \{h, SI, NO\}| - 1$. Etichettiamo gli elementi di Σ come $s_0 = \sqcup$, s_1, s_2, ..., s_v, ove $v = |\Sigma| - 1$. Ora raggruppiamoli in tre tipi di variabili booleane nel modo indicato dalla seguente tabella 2.1.

Conveniamo che una computazione di M induce un assegnamento di verità alle variabili esposte nella tabella (l'idea con cui si procede è descrivere le configurazioni della Macchina di Turing tramite una formula logica, che può essere anche lunghissima ma finita) e, che se M termina prima del tempo $p(n)$ la sua configurazione rimane immutata per tutti gli istanti successivi. D'altra parte, dato un arbitrario assegnamento alle variabili della tabella 2.1 non necessariamente questo corrisponde a una computazione effettiva di M. Ai fini della dimostrazione, la trasformazione f_L lavora costruendo un insie-

Tabella 2.1: SetsOfVariables

Variabile	Indici	Significato
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq r$	al termine dell'i-esimo passo della computazione, M è nello stato q_k
$H[i,j]$	$0 \leq i \leq p(n)$, $0 \leq j \leq p(n)$	al termine del passo i il cursore scandisce la casella j-esima
$S[i,j,k]$	$0 \leq i \leq p(n)$, $0 \leq j \leq p(n)$, $0 \leq k \leq v$	al termine del passo i-esimo la casella j contiene s_k

me di clausole C le cui variabili (quelle della tabella 2.1) sono tali che: un assegnamento di verità soddisfa l'espressione logica composta dalle clausole di C unite in FNC

se e solo se

è un assegnamento di verità indotto tramite f_L da una computazione di M che accetta un messaggio in ingresso x. Ovvero:

$$\begin{aligned} x \in L \quad &\Leftrightarrow \exists \text{ una computazione di } M \text{ che accetta } x \text{ messaggio in ingresso} \\ &\Leftrightarrow \exists \text{ un assegnamento di valori di verità per l'insieme di clausole } C \text{ unite in FNC indotto da } f_L \end{aligned}$$

Quindi f_L è effettivamente una trasformazione da L_M a L_{SAT}. Vediamo infine che è una riduzione in tempo polinomiale. Descriviamo ulteriormente f_L: suddividiamo le clausole di f_L in sei gruppi, raffigurati nella Tabella 2.2.

Questi servono a far corrispondere un assegnamento di valori di verità per un'espressione logica a una possibile computazione di M che termina in uno stato di accettazione del messaggio in ingresso. Il gruppo G_1 consiste delle clausole[3] 2.1 e 2.2:

[3]N.B.: gli elementi dell'insieme, i letterali, sono le particelle delle clausola e collegati da una disgiunzione, più clausole sono collegate da una congiunzione e formano l'espressione logica

Tabella 2.2: Restrizioni sugli assegnamenti di valori di verità

Gruppi di clausole	Restrizione imposta
G_1	al termine dell'i-esimo passo della computazione, M è in un dato stato
G_2	al termine del passo i il cursore scandisce una data casella
G_3	al termine del passo i-esimo ogni casella del nastro contiene un simbolo
G_4	al passo 0-esimo, la computazione è nella sua configurazione iniziale
G_5	entro il passo $p(n)$-esimo M accetta il messaggio in ingresso x
G_6	al termine del passo i-esimo, la configurazione di M al passo successivo è data dalla mappa δ applicata alla configurazione di M al tempo i

$$\{Q[i,0], Q[i,1], ..., Q[i,r]\} \text{ con } 0 \leq i \leq p(n) \tag{2.1}$$

$$\{\overline{Q[i,j]}, \overline{Q[i,j']}\} \text{ con } 0 \leq i \leq p(n), 0 \leq j \leq j' \leq r \tag{2.2}$$

Se consideriamo le clausole 2.1 tutte $p(n)+1$ insieme, otteniamo che possono essere soddisfatte se e solo se per ogni passo i, M si trova in almeno uno stato. Invece, per le $(p(n)+1)\cdot(r+1)\cdot(r/2)$ clausole 2.2, otteniamo che possono essere soddisfatte se e solo se non esiste un istante in cui M si trova in più di uno stato. Analogamente per i gruppi G_2 e G_3; e anche per G_4, G_5 consistenti di una clausola costituita di un solo letterale (vedi [2]). Notare che fino a qui il numero di clausole e di letterali è limitato superiormente da una funzione polinomiale in $p(n)$. Verifichiamo questo anche per l'ultimo gruppo di clausole G_6. Innanzitutto lo dividiamo in due sottogruppi: il

primo, corrispondente alle clausole in 2.3, che si occupano di garantire che il cursore non scandisce la casella j al passo i, quindi il simbolo in posizione j non cambia tra il passo i e quello successivo $i+1$.

$$\{\overline{S[i,j,l]}, H[i,j], S[i+1,j,l]\} \text{ con } (0,0,0) \leq (i,j,l) \leq (p(n),p(n),v) \quad (2.3)$$

Le $(p(n)+1)^2 \cdot (v+1)$ clausole 2.3 non possono essere soddisfatte se per l'istante i, casella j e simbolo s_l il cursore sul nastro non scansiona al passo i la casella j contenente s_l al passo i, ma non contenente s_l al successivo passo $i+1$; altrimenti sono soddisfatte per un opportuno assegnamento di valori di verità. Il secondo, corrispondente alle clausole in 2.4, si occupa di garantire che i cambiamenti da una configurazione alla successiva siano corretti rispetto alla definizione di mappa di transizione δ di M.

$$\begin{gathered}
\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, \overline{H[i+1,j+\Delta]}\} \\
\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, \overline{Q[i+1,k']}\} \\
\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, \overline{S[i+1,j,l']}\} \\
\text{con } (0,0,0,0) \leq (i,j,k,l) \leq (p(n),p(n),r,v) \text{ e} \\
\delta(q_k, s_l) \begin{cases} (q_{k'}, s_{l'}, D) & \text{, se } q_k \in Q \setminus \{SI, NO\} \\ (q_k, s_l, \bullet) & \text{, altrimenti} \end{cases}
\end{gathered} \quad (2.4)$$

Le $3 \cdot p(n) \cdot (p(n)+1) \cdot (r+1) \cdot (v+1)$ clausole 2.4, insieme con tutte quelle viste sopra, è facile vedere che permettono di costruire la mappa f_L tale che: dato $x \in L$ esiste una computazione in tempo polinomiale di M, Macchina di Turing non deterministica, a cui la mappa associa un assegnamento di valori di verità che soddisfa l'espressione logica consistente di tutte le clausole $C = G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5 \cup G_6$ messe in congiunzione tra loro. E viceversa dato un qualsiasi assegnamento di valori di verità che soddisfa un'espressione logica del tipo FNC di clausole di C, gli corrisponde tramite f_L una computazione di M terminante in uno stato di accettazione per il messaggio in ingresso x (per costruzione di f_L). Ovvero $f_L(x)$, istanza di SAT, è soddisfatta da un qualche assegnamento di valori di verità se e solo se $x \in L$. Infine f_L è stata costruita in tempo polinomiale in $n = |x|$. Infatti dato un linguaggio L e scelta una

Macchina di Turing non deterministica M in grado di riconoscerlo in tempo polinomiale $p(n)$, abbiamo definito per f_L l'insieme di variabili **U** contenute nella tabella 2.1 e di clausole C contenute nella tabella 2.2. Abbiamo che $|\mathbf{U}| = \boldsymbol{O}(p(n)^2)$ come si evince dalla tabella 2.1; e $|C| = \boldsymbol{O}(p(n)^2)$[4]. Nessuna clausola può contenere più di $2 \cdot |\mathbf{U}|$ letterali, e il numero di simboli richiesti per la notazione richiede solo di aggiungere un fattore $\log |\mathbf{U}|$, irrilevante nel considerare un limite superiore polinomiale. Quindi, espressa in funzione di $|\mathbf{C}| \cdot |\mathbf{U}|$, otteniamo una complessità dell'ordine di $\boldsymbol{O}(p(n)^4)$. Quindi la complessità dell'algoritmo per f_L è limitato superiormente da una quantità polinomiale in $|x|$ ($|x| \propto l[x]$, vedi [2]). □

2.2.2 L è contenuto in P

Concludiamo il Capitolo 2 con il risultato seguente:

Teorema 2.2.3. ***L*** $\subseteq$ ***P***

Dimostrazione. Una Macchina di Turing deterministica è un caso particolare di una non deterministica, se consideriamo quest'ultima con la condizione di una sola scelta nel cambio di configurazione da un passo al passo successivo della computazione. Questo vuol dire che **L** $\subseteq$ **NL**. Ora, vediamo che **NL** $\subseteq$ **P**. In virtù dell'Osservazione 3 bisogna che consideriamo una Macchina di Turing non deterministica N in grado di operare in spazio logaritmico (solo il messaggio in ingresso richiederebbe più di $\log n$ spazio). Supponiamo quindi che N abbia un primo nastro di sola lettura e un ultimo nastro di sola scrittura, in modo che le computazioni necessarie vengano effettuate su altri nastri e quindi non venga considerato lo spazio necessario per la sola lettura del messaggio in ingresso e la scrittura del messaggio in uscita, rimane così un numero $k - 2$ di nastri necessari alla computazione. Per ipotesi N deciderà L in spazio non deterministico $f(n) = \log n$, con $n = |x|$ lunghezza del messaggio in ingresso. Questa computazione possiamo simularla tramite una

[4] N.B.: r e v non compaiono perchè non dipendenti da $n = |x|$, e quindi contribuiscono solo di un fattore costante all'ammontare della complessità necessaria per costruire f_L

Macchina di Turing deterministica M in tempo polinomiale nel seguente modo: consideriamo il succedersi di configurazioni della computazione svolta da N. Ci basta conoscere la posizione i del cursore lungo il messaggio di ingresso, il suo stato q e il contenuto dei k nastri necessari alla computazione per esprimere un passo della computazione svolta da N. Il messaggio in ingresso e di uscita non sono rilevanti ai fini del contenuto della computazione svolta. Quindi, considerando $k-2$ stringhe scritte sui rimanenti $k-2$ nastri e lo stato q e la posizione i, possiamo unirli tutti in una k-tupla $(q, i, y_2, ..., y_{k-1})$. Allora abbiamo un numero di configurazioni date da $|K|$ scelte per la prima componente, $n+1$ scelte per i e al più $|\Sigma|^{f(n)\cdot(k-2)}$ scelte per le stringhe sui rimanenti nastri. Quindi riscrivendo, il numero totale di configurazioni di N assunte durante la computazione sul messaggio in ingresso x è al massimo $n \cdot c_1^{f(n)} = c_1^{\log n + f(n)}$ per una qualche costante c_1 dipendente da N. Considerato il grafico di vertici le possibili configurazioni di N e come archi il passaggio tra una computazione e la computazione successiva, dire $x \in L$ per N equivale a stabilire se esiste un percorso nel grafo appena costruito di nodo di partenza la configurazione iniziale $(s, 0, \triangleright, \underbrace{\epsilon, ..., \epsilon}_{k-2})$[5] e nodo di arrivo la configurazione in uno stato di terminazione $(SI, i, ...)$. Questo è un problema noto di esplorazione di grafi (i.e. REACHABILITY) risolubile in tempo polinomiale ($\boldsymbol{O}(n^2)$, con n dimensione del messaggio in ingresso) nella dimensione dell'input (i.e. il numero di vertici del grafo). Quindi possiamo simulare l'algoritmo non deterministico computato da N tramite un algoritmo deterministico operante su un grafo di dimensione $c_1^{\log n + f(n)}$ in tempo $\boldsymbol{O}(c_1^{2\cdot(\log n + f(n))}) = \boldsymbol{O}(n^4)$, dove il 4 è giustificato dal fatto che per ipotesi $f(n)$ è la quantità logaritmica di spazio usato da N operante su $n = |x|$. □

[5] N.B.: ϵ denota la stringa vuota, cioè una stringa senza caratteri

Capitolo 3

GI nel caso di alberi

Ora trattiamo un metodo per rispondere alla domanda se due dati alberi radicati abbiano in corrispondenza bigettiva i propri archi in modo che sia rispettata l'adiacenza di due vertici ad uno stesso arco. In generale, un albero è un grafo $G = (V, E)$ connesso tale che $\forall\ u, v \in V$ esista uno e un solo percorso che collega u a v in V. Dove un percorso è una sequenza di archi $(e_1, e_2, ...)$ che permette di passare attraverso i vertici da un vertice di partenza ad un vertice di arrivo. Per fare questo sono noti algoritmi che operano in **P** (vedi [7]), in **NC** (vedi [5] e [1]) e in **L** (vedi [5]). Vediamo nel dettaglio l'ultimo caso premettendo alcune nozioni preliminari.

3.1 Isomorfismo tra Alberi

Definizione 3.1. Siano S e T due alberi. Allora definiamo $S < T$ ricorsivamente nel seguente modo:

1. $|S| < |T|$[1] oppure,

2. $|S| = |T|$ e $\sharp s < \sharp t$[2] oppure,

[1] $|\cdot|$ indica la cardinalità dell'insieme dei vertici dell'albero $\cdot$

[2] $\sharp\,\cdot$ indica il numero di figli del vertice $\cdot$, con s e t radici dei relativi S e T

3. $|S| = |T|$ e $\sharp s = \sharp t = k$ e $(S[1], ..., S[k]) < (T[1], ..., T[k])$ lessicograficamente; ove gli interi fra quadre servono a indicare i sottoalberi figli della radice[3] e $S[1] \leq ... \leq S[k]$ e $T[1] \leq ... \leq T[k]$[4].

Osservazione 8. In generale, presi S e T tali che $S < T \nRightarrow$ ogni coppia di sottoalberi M ed N di S e T siano tali che $M < N$. Inoltre avere sottoalberi non isomorfi è condizione sufficiente per $\neg$ TI (TI stà per tree isomorphism).

Osservazione 9. La relazione $<$ appena definita è una relazione irriflessiva e transitiva (e quindi anche asimmetrica).

Osservazione 10. Consideriamo analogamente la relazione $\leq$ sull'insieme degli alberi, allora $\leq$ è una relazione di ordine totale.

Proposizione 3.1.1. Due alberi S e T sono isomorfi, nel senso esposto sopra di esistenza di una corrispondenza bigettiva fra i vertici che conserva gli archi, se e solo se né $S < T$ né $T < S$.

Dimostrazione. Siano k, h, i e j interi. Premettiamo che useremo con abuso di notazione k sia come $\sharp\, t$ che come indice sull'insieme $\{1, ..., \sharp\, t\}$ (idem per i); similmente sia $k_h := \sharp\, t_{k_{h-1}}$ se $h \geq 2$ e $k_h := \sharp\, t$ se $h = 1$ (e l'analogo per i_j). Supponiamo per assurdo che valga $S < T$, allora: se valgono le prime due di 3.1 esiste almeno un vertice in più (quello che rende valide 1 o 2) che non rispetta la condizione di isomorfismo; se invece vale la terza allora (per definizione di ordine lessicografico) possiamo trovare sottoalberi $S[k_1, ..., k_h]$ e $T[i_1, ..., i_j]$ per cui: $S[k_1, ..., k_h] < T[i_1, ..., i_j]$ oppure $S[k_1, ..., k_h] < T[i_1, ..., i_j]$ (in base alle prime 2 di 3.1 usate ricorsivamente). Assurdo per quanto visto sopra. Viceversa, costruiamo φ l'isomorfismo tra S e T. Si pone

$$\varphi(t) = s$$

[3] possiamo supporre per comodità che la mancanza di un intero (e.g. $S[\,]$) sottintenda il vertice radice di S, per un uso più generale della notazione vedi dimostrazione della proposizione 3.1.1

[4] Dati due alberi S e T definiamo $S = T$ se: *1-* $|S| = |T|$ o, *2-* $|S| = |T|$ e $\sharp\, t = \sharp\, s$ o, *3-* $|S| = |T|$, $\sharp\, t = \sharp\, s$ e $(S[1], ..., S[k]) = (T[1], ..., T[k])$ lessicograficamente

con t e s radici rispettivamente di S e T. Indichiamo come sopra $S[k_1, ..., k_h]$ ($T[i_1, ..., i_j]$) il sottoalbero di S (T) radicato nel k_h-esimo (i_j-esimo) figlio, che é figlio del k_{h-1}-esimo (i_{j-1}-esimo) vertice, e cosí via fino al k_1-esimo (i_1-esimo) vertice figlio della radice dell'albero. Siano le loro due radici $s[k_1, ..., k_h]$ ($t[i_1, ..., i_j]$). Definiamo

$$\varphi(s[k_1, ..., k_h]) = \varphi(t[i_1, ..., i_j])$$

ove per ipotesi né $(S_{1_h}, ..., S_{k_h}) < (T_{1_j}, ..., T_{i_j})$ lessicograficamente né $(T_{1_j}, ..., T_{i_j}) < (S_{1_h}, ..., S_{k_h})$ lessicograficamente. L'applicazione φ chiaramente non manda un arco o un percorso in un vertice, ergo è iniettiva. I vertici vengono considerati tutti per la 1 e la 2 della definizione ricorsiva 3.1 dato che dobbiamo avere la condizione che né $M < N$ né $N < M$, per ogni M, N sottoalberi di S, T. Infine $\varphi : S \to T$ conserva la relazione di adiacenza di coppie di vertici dato che associa ricorsivamente ad ogni radice $s[k_1, ..., k_h]$ i suoi figli nel sottoalbero $S[k_1, ..., k_h]$ in modo che in T si abbia la condizione né $(S_{1_h}, ..., S_{k_h}) < (T_{1_j}, ..., T_{i_j})$ lessicograficamente né $(T_{1_j}, ..., T_{i_j}) < (S_{1_h}, ..., S_{k_h})$ lessicograficamente. □

3.2 L'algoritmo

Adesso esponiamo come basandosi sulla definizione 3.1 si possa ottenere un algoritmo operante in **L** per risolvere TI. Consideriamo archi (u, v), ove u è padre di v, con un orientamento dato da etichette (i.e. un valore intero) associate ai vertici. L'orientamento è tale che l'etichetta di u è minore dell'etichetta di v. I figli di uno stesso vertice siano ordinati per valore decrescente delle etichette da sinistra verso destra. I valori interi usati sono gli interi da 1 a $n := |V|$. Così abbiamo un ordine (che possiamo ipotizzare essere computabile, vedi [5], in **L**) su tutto l'insieme dei vertici dell'albero. L'ordinamento è fondamentale per la sequenzializzazione dell'algoritmo. L'albero si suppone presentato nel nastro di sola lettura della Macchina di Turing eseguente l'algoritmo per TI, che supponiamo dotata anche di un nastro di

lavoro di lunghezza logaritmica nella dimensione del messaggio in ingresso. Il messaggio in uscita supponiamo venga scritto in un altro nastro dedicato allo scopo. Consideriamo le seguenti funzioni computabili ([5]) in **L**, impiegate per effettuare visite in profondità dell'albero in spazio logaritmico:

Genitore (u) la funzione restituisce il genitore del vertice nell'albero se esiste (i.e. il vertice t t.c. $(t, u) \in V$) e 0 altrimenti;

PrimoFiglio (u, ordine) la funzione restituisce il primo figlio di u se esiste (i.e. il vertice con la minore etichetta tra quelli di etichetta maggiore di quella di u, rispetto all'ordine passato come parametro alla funzione, tale che **Genitore (u)** ha come figlio u) e 0 se u non ha figli;

FiglioSuccessivo (u, ordine) la funzione restituisce il figlio successivo a u (i.e. il vertice, figlio di **Genitore (u)**, con la minore etichetta tra quelli di etichetta maggiore di quella di u), rispetto all'ordine, se esiste e 0 se u è il figlio maggiore.

Una visita in profondità viene effettuata memorizzando la posizione iniziale (ripassando per il vertice di partenza l'algoritmo termina) e l'ultima direzione percorsa (per evitare di retrocedere sullo stesso percorso). Nell'algoritmo di visita in profondità le direzioni sottostanno alle seguenti regole:

- All'inizio o l'ultima direzione presa è una discesa al **PrimoFiglio** (se esiste) o uno spostamento al **FiglioSuccessivo** (se esiste) $\Rightarrow$ si procede rispettivamente con: una discesa al **PrimoFiglio** (se esiste), uno spostamento al **FiglioSuccessivo** (se esiste) o una salita al **Genitore** (se esiste),
- altrimenti se l'ultima direzione presa è una salita (utilizzo della tecnica di backtracking) al **Genitore** (se esiste) $\Rightarrow$ si procede rispettivamente con: uno spostamento al **FiglioSuccessivo** (se esiste) o una salita al **Genitore** (se esiste),
- si termina la computazione quando ritorniamo al vertice di partenza.

Notiamo che l'etichetta di un nodo è un valore intero n non più grande di $|V|$, si ha che la memoria impiegata per la sua memorizzazione in qualche base b è $\log_b n$. Per la definizione 3.1 abbiamo quindi che $\sharp\ u$ si computa chiamando **PrimoFiglio (u, ordine)** una prima volta e successivamente chiamando **FiglioSuccessivo (u, ordine)** fino al valore di ritorno 0. Quindi il numero di chiamate che non ritornano 0 è $\sharp\ t$. Infine abbiamo che $|S|$, con S albero, si computa chiamando la procedura sopra descritta di visita in profondità partendo dal vertice radice dell'albero S e contando i nodi visitati con un contatore globale (inizializzato a 1) ogni volta che un nuovo nodo viene visitato. Di seguito si procede con la parte fondamentale dell'algoritmo, quella in cui determiniamo la relazione di preordine tra i due alberi dati S e T. Notiamo che per conseguenza del risparmio operato sullo spazio, l'algoritmo in generale richiederà una quantità di tempo più che lineare.
Siano ora dati quindi S e T due alberi (radicati rispettivamente in s e t), confrontiamo prima la loro dimensione (vedi Definizione 3.1, punto 1). Se sono uguali nella dimensione allora confrontiamo il numero di figli delle rispettive radici (vedi Definizione 3.1, punto 2). Se hanno lo stesso numero k di figli procediamo con il confronto lessicografico dei sottoalberi radicati nei k figli (vedi Definizione 3.1, punto 3). Chiaramente tra S e T l'albero più piccolo è quello col numero h (al più k) maggiore di sottoalberi di minor dimensione[5]; terminando ora nell'algoritmo non vi è bisogno di ulteriore ricorsione. Se h è uguale sia in S che in T allora procediamo analogamente con la successiva dimensione più piccola disponibile. Il procedimento continua finché non finiamo i sottoalberi figli (al più k) da confrontare. Se h non si è trovato uguale nel confronto di sottoalberi figli di S e di T abbiamo raggiunto uno stato di terminazione NO, altrimenti procediamo ricorsivamente come indicato di seguito. Notiamo che non essendosi usata fin'ora la ricorsione il grafico delle chiamate è aciclico. Quindi l'uso di una pila per le variabili locali e il contatore del programma (i.e. il puntatore alle istruzioni) risulta in un incremento dello spazio necessario alla memorizzazione di $\boldsymbol{O}(\log(n))$. Facciamo uso di

[5] dimensione è il valore $|S| = |V|$ con $S = (V, E)$, idem per T

una visita in profondità (in breve DFS) su due fronti (una visita in S e l'altra in T) e di confronti fra un sottoalbero figlio di S e un sottoalbero figlio di T[6], effettuando questa usando la dimensione dei rispettivi sottoalberi come ordine, e le etichette per non avere passaggi ambigui. Per effettuare il confronto incrociato fra due rispettivi sottoalberi figli di S e T si chiama ricorsivamente la procedura di confronto descritta finora passando dai sottoalberi radicati in (s, t), i due alberi stessi S e T, ai sottoalberi radicati nei vertici figli (s', t'), trovati mediante la doppia visita DFS. Queste chiamate ricorsive per il confronto incrociato avvengono all'interno di un gruppo di sottoalberi (uno in S e l'altro in T) della stessa dimensione (della stessa cardinalità) che d'ora innanzi chiameremo blocco. Procederemo di blocco in blocco, in ordine crescente a partire da quello riferito alla dimensione minima. Lavorando, con i confronti incrociati, tra blocchi della stessa dimensione via via maggiore. In la cui dimensione sarà ricalcolata al momento di ritornare la chiamata ai sottoalberi padri nella procedura di confronto incrociato, così da risparmiare spazio utilizzato. In questo modo, ritornando (s', t') ai rispettivi genitori, in modo da ritornare all'ambiente della chiamata ricorsiva precedente, non abbisognamo di memorizzare una pila di vertici passati come parametri alla chiamata. Consideriamo il confronto incrociato blocco per blocco a partire da quello di dimensione minore. Procediamo finché non troviamo una disuguaglianza secondo l'ordine lessicografico imposto ricorsivamente, oppure procedendo similmente nel blocco successivo. Osserviamo che se il numero di figli è k a questo punto della chiamata ricorsiva, dal fatto che i blocchi sono di sottoalberi equicardinali otteniamo che ogni chiamata ricorsiva ha dimensione del messaggio in ingresso al più $\frac{n}{k}$. Quindi abbiamo $\boldsymbol{O}(\log(k))$ bits nella pila della memoria locale, che ci permettono di usare un numero fissato di puntatori alle istruzioni variabile fra 1 e k (per i cicli). Chiaramente nel caso particolare di $k = 1$ non è necessario spazio aggiuntivo. Arriviamo quindi al punto che si presentano i casi seguenti:

$k = 0$ non ci sono figli nei due sottoalberi di S e T, allora essendo i vertici

[6] come da letteratura diciamo confronti incrociato.

tra loro isomorfi otteniamo due sottoalberi isomorfi (anche qui, non c'è nessuna chiamata ricorsiva),

$k = 1$ i sottoblocchi sono esattamente un blocco, S' e T' rispettivamente, notare che sono della stessa cardinalità. Siccome $|S'| + |T'| \leq n - 2$, non possiamo usare spazio aggiuntivo nella chiamata ricorsiva per il confronto tra i sottoalberi S' e T'. Con ciò, al momento di ritornare la chiamata, avendo memorizzato il punto di provenienza e avendo realizzato, con un'enumerazione, che abbiamo solo un sottoalbero di dimensione fissata, riconosciamo che la chiamata ricorsiva dell'algoritmo veniva proprio da qui. E quindi possiamo riprendere l'esecuzione nell'ambiente chiamante. Qualora si fosse ottenuto che S' e T' sono in relazione $<$ l'algoritmo termina con risposta NO. Altrimenti si prosegue per passare ai blocchi successivi di dimensione più grande,

$k \geq 2$ sia f una funzione tale che $V \to (\mathbb{N} \times \mathbb{N} \times \mathbb{N})$, e ad ogni vertice u, figlio della radice di un blocco, associa tre valori $f_1(u)$, $f_2(u)$ e $f_3(u)$ (il suo profilo), rispettivamente il numero di sottoalberi che nel blocco corrispondente hanno relazione $<$, $>$ e né $<$ né $>$ con il sottoalbero radicato in u (si ponga u a sinistra nel confronto tramite la relazione). Notare che f richiede al più $\boldsymbol{O}(\log(k))$ spazio per la memorizzazione delle tre componenti e per sequenzializzare il confronto incrociato, ma è necessario ripetere le computazioni più volte. Innanzitutto ($\diamond$) cerchiamo il sottoalbero nel blocco di k sottoalberi in S di valore massimo secondo la relazione $<$ (vedi Definizione 3.1) rispetto a quelli in T. Ovvero con il minimo valore di f_2 sull'insieme dei sottoalberi figli. Supponiamo per fissare le idee di trovarlo in S. Per cui avrà valore f_1 più grande tra quelli assunti dai sottoalberi figli. Questo implicherebbe immediatamente la terminazione con risposta NO dell'algoritmo per TI. Similmente il caso in T. Altrimenti, se troviamo un sottoalbero S' in S di valore minimo secondo la relazione $<$ (i.e. f_2 ha valore minimo), e lo stesso succede in T, allora né $S' < T'$ né $T' < S'$ per minimalità.

Nel caso ci trovassimo di fronte a più sottoalberi minimali dobbiamo confrontare i rispettivi valori di f_3, nel caso siano diversi abbiamo che l'algoritmo per TI termina nello stato NO. Altrimenti consideriamo S e T con lo stesso numero di sottoalberi minimali rispetto alla relazione $<$, allora il confronto riparte sui successivi sottoalberi più grandi, nello stesso blocco che abbiamo preso in considerazione. Notiamo che nel passo successivo il prossimo profilo f_1, f_2 e f_3 calcolato ha valore corrente $f_1 + f_2$ per il suo valore di f_1. Poniamo $h := f_3$, che è ben definito perché f_3 è uguale nei due blocchi in S e T arrivati a questo passo dell'algoritmo. Procedendo nel passaggio successivo si cercano ($\diamond$) i sottoalberi in S e T minimali, ignorando di una quantità h i valori in f_1, f_2 e f_3 di nuovo calcolati (per risparmiare sull'utilizzo di memoria). Così via si ripete il procedimento come sopra aggiornando ogni volta il valore di h, fino a raggiungere il valore k. Se raggiungiamo la fine, di tutto l'elenco di sottoalberi figli del blocco considerato, senza trovare una disuguglianza in ordine lessicografico nei profili (i.e. f_1, f_2 e f_3) in ogni blocco tra S e T, allora l'algoritmo per TI riparte di nuovo nel blocco successivo.

3.3 La complessità computazionale

Supponiamo che i k figli della radice nel caso di una chiamata ricorsiva nell'algoritmo precedente siano partizionati in gruppi, in modo che l'i-esimo gruppo sia di cardinalità k^i e i sottoalberi nell'i-esimo gruppo abbiano tutti dimensione N_i, con $N_1 \leq N_2 \leq ...$. Ovvero:

$$k^i := \{\sharp s^i | s^i \text{ sono radici dei sottoalberi figli, della}$$

$$\text{stessa dimensione } N_i, \text{ nell'}i\text{-esimo blocco }\}.$$

Data la relazione $\Sigma_i k^i \cdot |S^i| \leq n$ si ottiene $\frac{n}{k^i} \geq |S^i|$. Quindi l'algoritmo utilizza per TI una quantità di memoria descritta secondo la seguente relazione

ricorsiva:

$$\boldsymbol{S}(n) \leq \max_i(\boldsymbol{S}(\frac{n}{k^i}) + \log(k^i))$$

ove $k^i \geq 2$ per ogni i. È logaritmica in n in ordine di grandezza, per la seguente Osservazione 11:

Osservazione 11. Se la funzione $a(x)$, definita su $\mathbb{R}^+$, soddisfa la relazione

$$\begin{array}{lll} a(x) & \leq & a(\frac{x}{\beta}) + c \quad \text{, per } x > 1 \\ a(x) & \leq & 0 \qquad\qquad \text{, per } x \leq 1 \end{array}$$

con β numero naturale, allora $a(x)$ é $\boldsymbol{O}(\log_\beta x)$.

Dimostrazione. Notiamo che $a(x)$ è una funzione definita sui reali positivi, quindi $a(\frac{x}{\beta})$ è ben definita. Nel caso particolare di valori $N = \beta^n$ si ottiene, riscrivendo la relazione dell'enunciato, una successione tale che:

$$a_{\beta^n} \leq c \cdot \log_\beta N.$$

Più in generale, iterando la relazione fino alla condizione $x \leq 1$:

$$\begin{array}{lll} a(x) & \leq & c + a(\frac{x}{\beta}) \\ & \leq & c + c + a(\frac{x}{\beta^2}) \\ & \leq & c + c + c + a(\frac{x}{\beta^3}) \\ & \leq & ... \end{array} .$$

Dopo un numero $t = \lceil \log_\beta x \rceil$ di passaggi, otteniamo un termine $a(\frac{x}{\beta^t}) = 0$ e quindi

$$a(x) \leq c(1 + 1 + ... + 1) = (\lceil \log_\beta x \rceil + 1).$$

□

Bibliografia

[1] Computational Complexity, Christos H. Papadimitriou

[2] Computers and Intractability, M.R. Garey, D.S. Johnson

[3] The Computational Complexity, J. Torán

[4] The Complexity of Planar Graph Isomorphism, J. Torán e F. Wagner

[5] A Logspace Algorithm for Tree Canonization, S. Lindell

[6] An Introduction to the Analysis of Algorithms, R Sedgewick e P. Flajolet

[7] The Design and Analysis of Computer Algorithms, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman